Compositional modeling and transformation of multi-clocked mode automata

Christian Brunette, Jean-Pierre Talpin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Compositional modeling and transformation of multi-clocked mode automata

Christian Brunette , Jean-Pierre Talpin

N°5728

Octobre 2005

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5728--FR+ENG

Compositional modeling and transformation of multi-clocked mode automata

Christian Brunette* , Jean-Pierre Talpin†

Systèmes communicants
Projet Expresso

Rapport de recherche n ° 5728 — Octobre 2005 — 20 pages

Abstract: This article presents the modeling and implementation of multi-clocked mode automata using the model-driven engineering tool Gme(Generic Modeling Environment) and starting from a meta-model for the computer-aided embedded system design tool Polychrony.

The article presents the design of a meta-model in Gme for the data-flow multi-clocked synchronous formalism Signal of Polychrony, its extension to multi-clocked mode automata and the use of model transformation technologies of the Gme environment to embed the latter extension in the former workbench. The complete model and transformation process is formalized and given a formal operational semantics.

Key-words: Mode automata, model transformation, Gme, synchronous languages, Signal

(Résumé : tsvp)

* christian.brunette@irisa.fr
† jean-pierre.talpin@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Modélisation et transformation d'automate de mode multi-horloge

Résumé : Cet article présente le paradigme de conception réalisé pour modéliser des automates de mode multi-horloges dans l'environnment générique de modélisation (GME). Ce paradigme est construit comme une extension de Signal-Meta, le paradigme réalisé pour le formalisme synchrone orienté flot de données de la plateforme POLYCHRONY, SIGNAL.

L'article décrit rapidement le métamodèle Signal-Meta, et plus précisément l'extension faite de ce métamodèle pour représenter les automates de mode et la manière dont de tels automates sont traduits en SIGNAL. Une sémantique formelle est donnée pour tous les mo'èles et les transformations qui leur sont appliquées.

Mots-clé : Automata de mode, transformation de modèle, GME, langages synchrones, SIGNAL

Contents

1 Introduction

Inspired by concepts and practices borrowed to digital circuit design and automatic control, the *synchronous hypothesis* has been proposed in the late '80s and extensively used for embedded software design ever since to facilitate the specification and analysis of control-dominated systems. Nowadays synchronous languages are commonly used in the European industry, especially in avionics, to rapidly prototype, simulate, verify and synthesize embedded software for mission critical applications.

In this spirit, synchronous data-flow programming languages, such as LUSTRE [1], Lucid [12] and SIGNAL [4], implement a model of computation in which time is abstracted by symbolic synchronization and scheduling relations to facilitate behavioral reasoning and functional correctness verification. While block diagrammatic modeling concepts are best suited for data-flow dominated applications, control-dominated processes may sometimes be preferably modeled using imperative formalisms, such as Esterel [17], Statecharts [15] or Syncharts [16].

In the particular case of the POLYCHRONY workbench, on which SIGNAL is based, time is represented by *partially ordered* synchronization and scheduling relations, to provide an additional ability to model high-level abstractions of system paced by multiple clocks: globally asynchronous systems. This gives the opportunity to seamlessly model heterogeneous and complex distributed embedded systems at a high level of abstraction, while reasoning within a simple and formally defined mathematical model.

In POLYCHRONY, design proceeds in a compositional and refinement-based manner by first considering a weakly timed data-flow model of the system under consideration and then provide expressive timing relation to gradually refine its synchronization and scheduling structure to finally check correctness of the assembled components using assumption/guarantee reasoning. SIGNAL favors the progressive design of correct by construction systems by means of well-defined model transformations, that preserve the intended semantics of early requirement specifications to eventually provide a functionally correct deployment on the target architecture of choice.

Previous work Gathering advantages of declarative and imperative approaches, mode automata were originally proposed by Maraninchi et al. [3] to extend the functionality-oriented data-flow paradigm with the capability to model transition systems easily and provide an additional imperative flavor. Similar variants and extensions of the same approach to mix multiple programming paradigms or heterogeneous models of computation [13] have been proposed until recently, the latest advance being the combination of stream functions with automata in [14]. Nowadays, commercial toolsets such as the Esterel Studio's Scade or Matlab/Simulink's Stateflow are largely inspired from similar concepts.

Contributions While the introduction of preemption mechanism in the multi-clocked data-flow formalism Signal was previously studied by Rutten et al. in [19], no attempt has been made to extend mode automata with the capability to model multi-clocked systems

and multi-rate systems, which is the aim of this article. Taking advantage of recent works extending POLYCHRONY with a meta-model [18] in the model-driven engineering framework of GME (Generic modeling environment [2]), we position our problem as extending to the meta-model on which SIGNAL is based with an inherited meta-model of multi-clocked mode automata to finally demonstrate how the later can be translated in the former by operating a model transformation. We put an emphasis on simplicity both for the specification (one third of a page, Figure 1) and for the formalization (five rules, Section 5.1) of mode automata.

Implementation This framework of mode automata was specified and implemented in the matter of a month, thanks to the facilities offered by the GME environment and is currently being ported on Eclipse [10].

Roadmap Section 2 first gives an informal presentation of the SIGNAL formalism, and of the timed data-flow graphs structure it is based on, before to outline the extension with mode we propose. Section 3 then presents the meta-model on which SIGNAL is based (a meta-model for timed data-flow graphs) and its extension with modes. Section 4 provides an operational framework to specify the semantics of data-flow graphs and mode automata based on micro-step synchronous automata [11]. Section 5 gives the specification for the translation of mode automata by data-flow graphs before addressing possible variants and extensions.

2 Data-flow specification of multi-clocked systems

We position the problem by considering multi-clocked synchronous (or polychronous) specifications using the data-flow formalism SIGNAL [4].

2.1 Multi-clocked data-flow

A SIGNAL process consists of the simultaneous composition of equations on signals that partially relate them with respect to an abstract timing model. In SIGNAL [4], a process p is an infinite loop that consists of the synchronous composition $p \,|\, q$ of simultaneous equations $x = y\, f\, z$ over signals noted x, y, z. Restricting the lexical scope of a signal name x to a process p is noted p/x.

$$p, q \;::=\; p \,|\, q \mid p/x \mid x = \overbrace{y\, f\, z}^{e} \qquad \text{(process)}$$

An equation $x = y\, f\, z$ defines partially timed relations between an expression e over input signals and an output signal. There are three primitive expressions in SIGNAL: delay, sampling and merge.

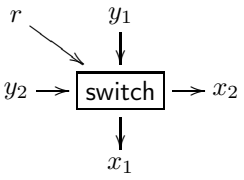
- A delay equation $x = y \,\mathsf{pre}\, v$ initially defines the signal x by the value v and then by the previous value of the signal y. In a delay equation, the signals x and y are assumed to be synchronous, i.e., either simultaneously present or simultaneously absent at all times.

- A sampling $x = y \,\mathsf{when}\, z$ defines x by y when z is true and both y and z are present. In a sampling equation, the output signal x is present iff both input signals y and z are present and z holds the value true.

- A merge $x = y \,\mathsf{default}\, z$ defines x by y when y is present and by z otherwise. In a merge equation, the output signal is present iff either of the input signals y or z is present.

2.2 Example of a crossbar switch

To support the presentation of our modeling techniques thorough the article, we consider the example of a simple crossbar switch. Its interface is composed of two input data signals y_1 and y_2 and a reset input signal r. Data signals are routed along the output data signals x_1 and x_2 upon the internal state s of the switch. The state is toggled using the reset signal by the functionality $\mathsf{toggle}(s, r)$. Data is routed along an output signal x from two possible input sources y_1 or y_2 upon depending on the value of s by two instance of the functionality $x = \mathsf{route}(s, y_1, y_2)$.



$$(x_1, x_2) = \mathsf{switch}(y_1, y_2, r) \quad \stackrel{\text{def}}{=} \quad \left(\begin{array}{rcl} s & = & \mathsf{toggle}(r) \\ \mid x_1 & = & \mathsf{route}(s, y_1, y_2) \\ \mid x_2 & = & \mathsf{route}(s, y_2, y_1) \end{array} \right) /s$$

The subprocess toggle defines the state of the switch by the signal s. If the reset signal r is present and true, then the next state t is defined by the negation of current state s and otherwise by s.

$$s = \mathsf{toggle}(r) \quad \stackrel{\text{def}}{=} \quad (s = t \,\mathsf{pre}\, \mathsf{true} \mid t = \mathsf{not}\, s \,\mathsf{when}\, r \,\mathsf{default}\, s) \,/t$$

The subprocess route selects which of the values v and w of its input signals y or z to send along its output signal x depending on the boolean signal s. If s is present and true, it chooses v and else, if s is present and false, it chooses w.

$$x = \mathsf{route}(s, y, z) \quad \stackrel{\text{def}}{=} \quad x = (y \,\mathsf{when}\, s) \,\mathsf{default}\, (z \,\mathsf{when}\, \mathsf{not}\, s)$$

Remember that SIGNAL equations partially synchronize input and output signals. In the route process, this implies that none of the signals y, z and s are synchronized, and that the output signal x is present iff either y is present and s true or z is present and s false.

2.3 Modeling multi-clocked systems

Just as in the POLYCHRONY workbench it is built upon, the data-flow synchronous formalism SIGNAL supports an intermediate representation of multi-clocked specification that exposes its control and data-flow properties for the purpose of analysis and transformation. A process p is represented as a data-flow graph G. In this graph, a vertex g is a data-flow relation

that partially defines a clock or a signal. A signal vertex $c \Rightarrow x = f(y_{1..n})$ partially defines x by $f(y_{1..n})$ at the clock c. A clock vertex $\hat{x} = e$ defines a relation between two particular signals or events called clocks.

$$G, H ::= g \,|\, (G \,\|\, H) \,|\, G/x \qquad \text{(graph)} \qquad g, h ::= \hat{x} = e \,|\, c \Rightarrow x = f(y_{1..n}) \qquad \text{(vertices)}$$

A clock c expresses control and defines a condition upon which a data-flow relation is executed. The clock $\hat{x}$ defines when the signal x is present (its value is available). The clocks x and $\neg x$ mean that x is true and false, respectively, and hence present. A clock expression e is Boolean expression that defines how a clock is computed. 0 means never.

$$c ::= \hat{x} \,|\, x \,|\, \neg x \qquad \text{(clock)} \qquad e ::= 0 \,|\, c \,|\, e_1 \setminus e_2, \,|\, e_1 \vee e_2 \,|\, e_1 \wedge e_2 \qquad \text{(expression)}$$

The decomposition of a process into the synchronous composition of clock and signal vertices is defined by induction on the structure of p. Each equation is decomposed into data-flow functions guarded by a condition, the clock $\hat{x}$ of the output. This clock will need to be computed for the function to be executed.

$$\begin{aligned}
G_{[x=y \text{ pre } v]} &\stackrel{\text{def}}{=} (\hat{x} \Rightarrow x = y \text{ pre } v) \,\|\, (\hat{x} = \hat{y}) \\
G_{[x=y \text{ when } z]} &\stackrel{\text{def}}{=} (\hat{x} \Rightarrow x = y) \,\|\, (\hat{x} = \hat{y} \wedge z) \\
G_{[x=y \text{ default } z]} &\stackrel{\text{def}}{=} (\hat{y} \Rightarrow x = y) \,\|\, (\hat{z} \setminus \hat{y} \Rightarrow x = z) \,\|\, (\hat{x} = \hat{y} \vee \hat{z}) \\
G_{[p \,|\, q]} &\stackrel{\text{def}}{=} G_{[p]} \,\|\, G_{[q]} \\
G_{[p/x]} &\stackrel{\text{def}}{=} G_{[p]}/x
\end{aligned}$$

Case of the crossbar switch Let us construct the graph of the crossbar switch. It can modularly be defined by one instance of the toggle functionality and two instances of the router. Each functionality can be decomposed into an untimed data-flow graph and its specific timing model be expressed by clock relations.

$$\begin{aligned}
G_{\text{switch}} &\stackrel{\text{def}}{=} (G_{\text{toggle}} \,\|\, G_{\text{route}_1} \,\|\, G_{\text{route}_2}) \,/ st \\
G_{\text{toggle}} &\stackrel{\text{def}}{=} (\hat{s} \Rightarrow s = t \text{ pre true}) \,\|\, (r \Rightarrow t = \text{not } s) \,\|\, (\hat{t} \setminus r \Rightarrow t = s) \,\|\, (\hat{t} = \hat{s}) \\
0 < i \neq j \leq 2,\ G_{\text{route}_i} &\stackrel{\text{def}}{=} (s \Rightarrow x_i = y_i) \,\|\, (\neg s \Rightarrow x_i = y_j)
\end{aligned}$$

2.4 Mode automata

The switch is a typical example of specification where an imperative automata-like structure superimposed to a native data-flow structure gives a shorter and more intuitive description of the system's behavior. The mode automata of the switch consists of two states flip and flop, in which routing is performed from $y_{1,2}$ to either $x_{1,2}$ or $x_{2,1}$ depending on the current mode of the automaton. The mode toggles from flip to flop, or converse, upon an occurrence of the event r.

$$(x_1, x_2) = \mathsf{switch}(y_1, y_2, r) \quad \stackrel{\text{def}}{=} \quad \left(\begin{array}{ll} \mathsf{init} & \mathsf{flip} : (x_1 = y_1 \,|\, x_2 = y_2) \\ \| & \mathsf{flop} : (x_1 = y_2 \,|\, x_2 = y_1) \\ \| & r \Rightarrow \mathsf{flip} \rightarrow \mathsf{flop} \\ \| & r \Rightarrow \mathsf{flop} \rightarrow \mathsf{flip} \end{array} \right)$$

To express mode automata, we consider a super-class of our meta-model for SIGNAL, which comprises the following base syntactic elements. $\mathsf{init}\, s$ specifies the initial state of an automaton a, $s : p$ the behavior p associated to mode s and $e \Rightarrow s \rightarrow t$ gives the clock e (or guard) of the transition from state s to state t. Automata a and b naturally compose synchronously with $a \,|\, b$.

$$a, b ::= \mathsf{init}\, s \mid (s : p) \mid (e \Rightarrow s \rightarrow t) \mid a \,|\, b$$

3 A meta-modeling approach

To develop our meta-modeling approach, we have used the GME environment [2]. GME is a configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments. It is developed by the ISIS institute at Vanderbilt University, and freely available at [8]. Meta-models are proposed in GME to describe *modeling paradigms* for specific domains. Such a paradigm includes, for a given domain, the necessary basic concepts in order to represent models from a syntactical viewpoint to a semantical one.

3.1 The SIGNAL meta-model

The definition of a meta-model in GME is realized using a specific paradigm called *MetaGME*. First, modeling paradigm concepts are described in an UML class diagram. To achieve it, MetaGME offers some predefined UML-stereotypes [9], among which *FCO*, *Atom*, *Model*, and *Connection*. FCO (*First Class Object*) constitutes the basic stereotype in the sense that all the other stereotypes inherit from it. It is used for expressing abstract concepts. Atoms are elementary objects that cannot include any sub-part. On the contrary, Models may be composed of several FCOs. Containment and Inheritance relations are represented as in UML. All the other types of relations are specified through Connections. Some of these stereotypes are used in the class diagram in Fig. 1. For the SIGNAL meta-model (Signal-Meta), class diagrams describe as concepts all syntactic elements defined in SIGNAL v4 [7]. Among these concept, there are an Atom for each SIGNAL operator (e.g. numeric, clock relations, constraints), a Model for each SIGNAL *container* (e.g. process declaration, module), and a Connection for each relation between SIGNAL operators (e.g. definition, dependence).

In these class diagrams, GME provides a means to express the visibility of FCOs within a Model through the notion of *Aspect*. Each FCO must be associated, at least, to one Aspect in which it is visible. Signal-Meta comprises two main Aspects: *Computation part* and *Clock and Dependence Relations*. The first Aspect manages all data-flow relations of the model, and the second one, all clock relations between signals.

The last step of the definition of a modeling paradigm in GME is to add some *OCL Constraints* in order to check some dynamic properties on a model designed with this paradigm. In Signal-Meta, OCL constraints check the validity of attribute values, such as the uniqueness of names inside a Model.

3.2 Refinement of the meta-model with modes

To manage mode automata, we extend Signal-Meta with a new class diagram represented in Fig. 1. An Automaton is a Model composed of states, transitions, and *StateObservers*.

There are three kinds of state: *AndState*, *Automaton*, and *State*. The two former are Model composed of other states, whereas the last one is a terminal state composed of equations. An AndState consists of two or more states that are composed in parallel. An Automaton can be added to another Automaton as a state, or to one of the Signal-Meta Models, represented in the class diagram by the *ModelsWithDataflow* Model. Thus, mode automata can be easily compose with SIGNAL programs or with other mode automata. This abstract concept represents Models including the two Aspects mentioned in previous section and all operators described in Signal-Meta. State inherits from this Model to be able to describe SIGNAL equations. Finally, the *InitState* Atom is dedicated to be connected to the initial state of the automaton.

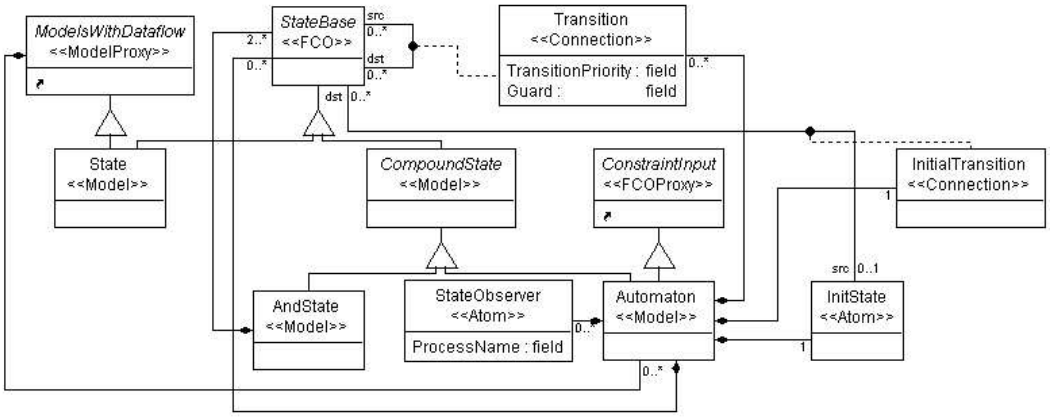


Figure 1: Extension of the SIGNAL meta-model for mode automata.

Transitions are represented as Connection in the meta-model. There are two kinds of transitions: *InitialTransition*, and *Transition*. InitialTransition connects an InitState Atom to a state of an Automaton. There can only be one such Connection in an Automaton. For a Transition, it connects a state to another. The source and the destination can be the same state. This connection has a *Guard* attribute, which corresponds to a boolean expression. To guarantee the determinism of the automaton, an attribute (*TransitionPriority*) is added to express the priority of a Transition. Thus, the smaller the value of the transition is, the

more the transition has priority. An OCL constraint checks that for each state, all output transitions have different priorities.

To follow the state of an automaton, we add the *StateObserver* Atom, which allows to call a process using the current state of the automaton as the only input signal of this process. The name of the process is given through the attribute *ProcessName*. If this attribute is not defined, the current state is written on the standard output.

Basically, the clock of an automaton depends of the clock of the signals used in all its transitions and states. The clock of an automaton can alternatively be explicitly specified. In the meta-model, this is translated by the inheritance of *Automaton* from *ConstraintInput*.

3.3 Example of the switch

We illustrate the use of the mode automata extension on the example of the switch. Fig. 2 represents the mode automaton of the switch inside the GME environment. Concepts described in the previous section appear in the bottom frame of Fig. 2.

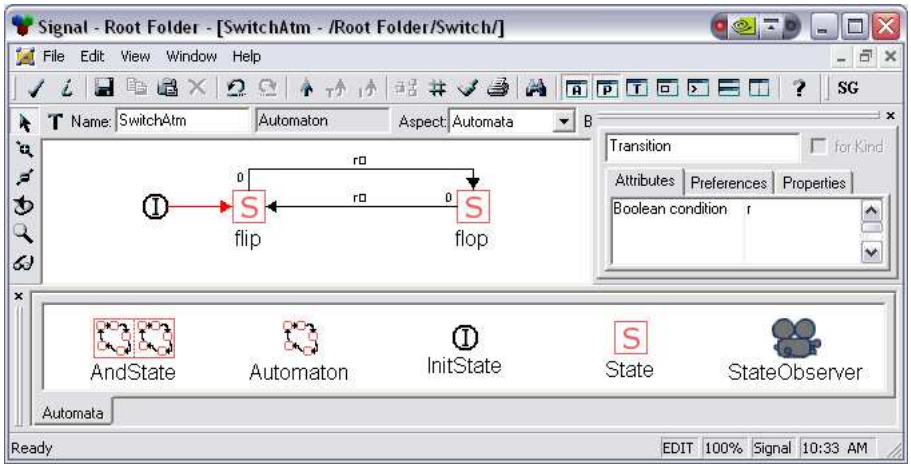


Figure 2: Example of the `SwitchAtm` automaton in GME.

The `SwitchAtm` automaton contains two terminal states (`flip` and `flop`). Transitions are guarded by the value of the event `r`, as labeled on the middle of transitions. The 0 indicates the transition priority. The content of `flip` (resp. `flop`) state is represented at the left (resp. right) of Fig. 3. On these figures, dotted arrows correspond to partial definitions in SIGNAL, and `x1`, `x2`, `y1`, and `y2` are references to signal declared in an upper Model.

The upper Model is that of the `switch` and the `SwitchAtm` automaton and all the signals it uses are declared there. Fig. 4 shows two main Aspects of the `switch` Model: the computation part on the left, and the clock relation on the right. In this Model, `y1`, `y2`, and `r` are input signals, and `x1` and `x2` are output signals.

Figure 3: Content of states `flip` and `flop`.

In the clock relation Aspect, the clock of `SwitchAtm` is synchronized with the union of the clock of the `y1`, `y2`, and `r`. The clock of `x1` and `x2` have to be specified explicitly because they are defined using partial definition. Therefore, the *MinClock* operator is used to define the clock of `x1` and `x2` as the union of clocks of their partial definitions. The `DATA_TYPE` parameter is only used to define a generic type for input and output signals.

Figure 4: The `switch` process.

4 Operational semantics framework

The semantics of multi-clocked data-flow graphs and mode automata is described by considering the theory of synchronous micro-step automata proposed by Potop et al. in [5]. As already demonstrated for SIGNAL, this framework accurately renders concurrency and causality for synchronous (multi-clocked) specifications [6].

4.1 Micro-step synchronous automata

Micro-step automata communicate through signals $x \in X$. The *labels* $l \in L_X$ generated by the set of names X are represented by a partial map of *domain* from a set of signals X noted $\text{vars}(l)$ to a set of values $V^\perp = V \cup \{\perp\}$ and tags. The label $\perp$ denotes the *absence* of communication during a transition of the automaton. We note $l' \leq l$ iff there exists l'' disjoint from l' such that $l = l' \cup l''$ and then $l \setminus l' = l''$. We say that l and l' are *compatible*, written $l \bowtie l'$, iff $l(x) = l'(x)$ for all $x \in \text{vars}(l) \cap \text{vars}(l')$ and, if so, note $l \cup l'$ their union. We write $\text{supp}(l) = \{x \in X \,|\, l(x) \neq \perp\}$ for the *support* of a label l and $\perp_X$ for the empty support.

Synchronous automata account for primitive communications using read and write operations on*directed communication channels* pairing variables x with directions represented by tags. Emitting a value v along a channel x is written $!x = v$ and receiving it $?x = v$. We write $\text{vars}(D)$ for the channel names associated to a set of directed channels D. The undirected or untagged variables of a synchronous automaton are its *clocks* noted c.

An *automaton* $A = (s^0, S, X, \rightarrow)$ is defined by an initial state s^0, a finite set of states S noted s or $x = v$, labels L_X and by a transition relation $\rightarrow$ on $S \times L_X \times S$. The *product* $A_1 \otimes A_2$ of $A_i = (s_i^0, S_i, X_i, \rightarrow_i)$ for $0 < i \leq 2$ is defined by $((s_1^0, s_2^0), S_1 \times S_2, X_1 \cup X_2, \rightarrow)$ where $(s_1, s_2) \rightarrow^l (s_1', s_2')$ iff $s_i \rightarrow^{l|_{X_i}} s_i'$ for $0 < i \leq 2$ and $l|_{X_i}$ the projection of l on X_i. An automaton $A = (s^0, S, X, \rightarrow)$ is *concurrent* iff $s \rightarrow^{\perp} s$ for all $s \in S$ and if $s \rightarrow^l s'$ and $l' \leq l$ then there exists $s'' \in S$ such that $s \rightarrow^{l'} s''$ and $s'' \rightarrow^{l \setminus l'} s'$. A *synchronous* automaton $A = (s^0, S, X, c, \rightarrow)$, of clock $c \in X$, consists of a concurrent automaton $(s^0, S, X, \rightarrow)$ which satisfies

- 1. $s \rightarrow^l s$ implies $l = c$ or $c \not\leq l$
- 2. $s^0 \rightarrow^c s^0$
- 3. $s \rightarrow^c s'$ implies $s' \rightarrow^c s'$
- 4. if $s_{i-1} \rightarrow^{l_i} s_i$ and $l_i \neq 1$ for $0 < i, j \leq n$ then $\mathrm{vars}(l_i) \cap \mathrm{vars}(l_j) = \emptyset$ iff $i \neq j$.

We assume that a channel x connects at most one emitter with at most one receiver. Multicast will however be used in examples and is modeled by substituting variable names (one $!x = v$ and two $?x = w_{1,2}$ will be substituted by two $!x = v$, $!x_2 = v$ and two $?x = w_1$ $?x_2 = w_2$ by introducing a local signal x_2).

The composition of automata is defined by synchronized product and synchronous communication using 1-place synchronous FIFO buffers. The synchronous FIFO of clock c and channel x is noted $\textsc{sfifo}(x, c)$. It serializes the emission event $!x = v$ followed by the receipt event $?x = v$ within the same transition (the clock tick c occurs after).

$$\textsc{sfifo}(x,c) \quad \stackrel{\text{def}}{=} \quad \left(s_0, \{s_{0..2}\}, \{?x, !x, c\}, c, \; \begin{array}{c} c \circlearrowright s_0 \xrightarrow{!x=v} s_1 \xrightarrow{?x=v} s_2 \\ \qquad\qquad c \end{array} \right)$$

Two synchronous automata are *composable* if their tagged variables are mutually disjoint. Let $A_i = (s_i^0, S_i, X_i, c_i, \rightarrow_i)_{i=1,2}$ be two composable synchronous automata and c a clock and write $A[c_2/c_1]$ for the substitution of c_1 by c_2 in A. The synchronous composition $A_1 \,|^c A_2$ is defined by the product of A_1, of A_2 and of a series of synchronous FIFO buffers $\textsc{sfifo}(x, c)$ that are all synchronized at the same clock c.

$$A_1 \,|^c A_2 \quad \stackrel{\text{def}}{=} \quad (A_1[c/c_1]) \otimes \left(\bigotimes_{x \in \mathrm{vars}(X_1)}^{x \in \mathrm{vars}(X_2)} \textsc{sfifo}(x, c) \right) \otimes (A_2[c/c_2])$$

4.2 Micro-step semantics of SIGNAL's data-flow graphs

Micro-step automata provide a simple and expressive operational framework to formalize the semantics of multi-clocked data-flow graphs.

Clocks A clock expression e corresponds to a transition system T_e^{st} from s to t which evaluates the presence of signals in accordance to e.

$$T_c^{s,t} \stackrel{\text{def}}{=} \left(s \xrightarrow{l_c} t\right) \qquad T_{c\wedge d}^{s,t} \stackrel{\text{def}}{=} \left(\begin{array}{ccccc} & & s' & & \\ & \overset{l_c}{\nearrow} & & \overset{l_d}{\searrow} & \\ s & & \xrightarrow{l_c l_d} & & t \\ & \underset{l_d}{\searrow} & & \underset{l_c}{\nearrow} & \\ & & t' & & \end{array} \right) /s't' \qquad T_{c\vee d}^{s,t} \stackrel{\text{def}}{=} \left(T_{c\wedge d}^{st} \cup T_c^{st} \cup T_d^{st}\right)$$

We write l_c for the label l that corresponds to the clock c and canonically denote v_x the generic value of the signal x.

$$l_{\hat{x}} \stackrel{\text{def}}{=} (?x = v_x) \qquad l_x \stackrel{\text{def}}{=} (?x = 1) \qquad l_{\neg x} \stackrel{\text{def}}{=} (?x = 0)$$

Relations A synchronization relation $\hat{x} = e$ accepts the events $\hat{x}$ and e in any order, or none of them, and then performs a clock transition c. Hence, the conditions expressed by $\hat{x}$ and e need to occur at the same time.

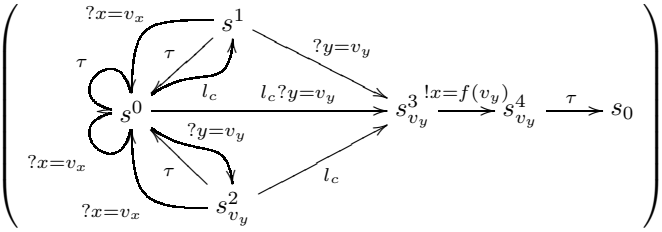
$$A_{\hat{x}=e} \stackrel{\text{def}}{=} \left(s, \{s,t\}, \{c,x\} \cup \text{vars}(e), c, \left(t \xrightarrow{c} s\right) \bigcup_{v_y \in V \mid y \in \text{vars}(e)}^{v_x \in V} T_{\hat{x}\wedge e}^{s,t}\right)$$

Clock expressions must be rewritten to fit the definition of T_e:

$$\begin{array}{rcl} \hat{x} = e \wedge f & \stackrel{\text{def}}{=} & (\hat{x} = \hat{y} \wedge \hat{z} \,|\, \hat{y} = e \,|\, \hat{z} = f) \,/yz \\ \hat{x} = e \vee f & \stackrel{\text{def}}{=} & (\hat{x} = \hat{y} \vee \hat{z} \,|\, \hat{y} = e \,|\, \hat{z} = f) \,/yz \\ \hat{x} = e \setminus f & \stackrel{\text{def}}{=} & (\hat{x} = y \,|\, \hat{y} = e \vee f \,|\, \neg y = f) \,/y \end{array}$$

Equations A partial equation $c \Rightarrow x = f(y)$ synchronizes x to the value of f by y at the clock c. But x may also be present when either c or y is absent. Therefore, the automaton requires x to be emitted with the value $f(v_y)$ only after the events y and c have occurred. If at least one of either c or y is present, then x may or may not be present with some value u computed by another partial equation. The semantics (combinatorially) generalizes to the case of $c \Rightarrow x = f(y_{1..n})$ with $n \geq 0$.

$$A_{c\Rightarrow x=f(y)} \stackrel{\text{def}}{=} \left(\begin{array}{l} s^0, \{s^{0..1}, s_{v_y}^{2..4}, \mid v_y \in V\}, \{x,y\} \cup \text{vars}(d), \tau, \\ \quad \bigcup_{v_x, v_y \in V}^{v_z \in V \mid z \in \text{vars}(c)} \left(\begin{array}{l} \\ \\ \end{array} \right. \end{array} \right.$$



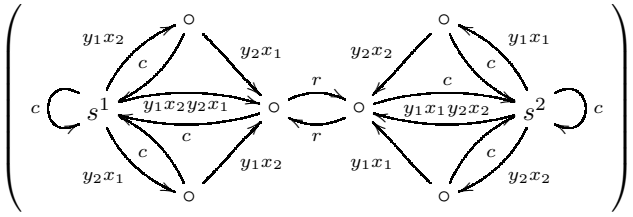
$$\left. \left. \begin{array}{l} \\ \\ \end{array} \right) \right)$$

Structuration *Composition $p \| q$ and restriction p/x* are defined by structural induction starting from the previous axioms with

$$A_{p\,|\,q} \quad \stackrel{\text{def}}{=} \quad A_p \|^c A_q \qquad A_{p/x} \quad \stackrel{\text{def}}{=} \quad (A_p)/x$$

Example 1. Consider the transition system for the switch process (the notation $y_j x_i$ stands for two steps $\circ \rightarrow^{?y_j=v} \circ \rightarrow^{!x_j=v} \circ$). The switch automaton consists of two mirrored structures that allow for concurrently receiving y_1 and y_2 and transmitting them along x_1 or x_2 according to the mode s_1 or s_2, toggled using the signal r.

$T_{\mathsf{switch}} \quad =$



4.3 Operational semantics of mode automata

The operational semantics of a mode automaton is described using one equation to define the micro-step automaton A_a corresponding to the mode declaration a. To this end, a mode automaton a is considered as a set of synchronously composed modes and transitions. Hence, we write $(s : p) \in a$ and $(c \Rightarrow s \rightarrow t) \in a$ for the modes and transitions it contains.

The semantics of a mode automaton a consists of a transition system that is the union of the transition systems of all modes $(s : p) \in a$. The transition system of a mode $(s : p)$ consists of T_p (that of the process p) where s_p (the initial state) is substituted by s (the mode state). For all mode transitions $c \Rightarrow s \rightarrow t \in a$, the transition system is completed with the transitions from the final states u of T_p to the mode state t.

We write $\text{final}(T) = \{s \,|\, t \rightarrow^\tau s \in T\}$ for the final states of T (the sources of clock transitions τ in T) and $S_a = \{s \,|\, (s : p) \in a\}$ for the states of a. As usual, s_a denotes the initial state of a and, referring to automaton A_p of a process p, s_p its initial state, S_p its states, X_p its variables and T_p its transition system.

$$A_a \quad \stackrel{\text{def}}{=} \quad \left(s_a, S_a \bigcup_{(s:p)\in a} S_p, \text{vars}(a), \bigcup_{(s:p)\in a} \left(T_p[s/s_p] \bigcup_{u\in \text{final}(T_p)\setminus s_p}^{(c\Rightarrow s\rightarrow t)\in a} \left(\left(A_c^{ur} \cup r \xrightarrow{\tau} t \right) /r \right) \right) \right)$$

Example 2. In the case of the switch, this amount to superimpose two transitions of condition r to the transition systems of the flip and flop modes.

$$T_{\mathsf{switch}} \quad = \quad \left(T_{\mathsf{flip}} \circ \underset{r}{\overset{r}{\rightleftarrows}} \circ T_{\mathsf{flop}} \right)$$

5 Model transformation

We use the POLYCHRONY workbench on top of which our modeling framework is built to perform formal verification (model checking and controller synthesis are provided with the Sigali tool) and code generation (in C, C++ or Java) starting from mode automata. Taking advantage of the meta-modeling framework provided by the GME, we define this necessary mapping from the meta-model of mode automata to that of the timed data-flow graphs of POLYCHRONY.

5.1 Compilation of mode automata

The compilation of a mode automaton as a timed data-flow graph consists of its structural translation into partial equations modeling guarded commands and the necessary synchronization relations described by clock equations. The top level rule C_a defines the current state of a, represented by a signal x (its next value being synchronously carried by the x'). The clock of the mode automaton is hence $\hat{x}$. It is synchronized to the clock expression e_x, the activity clock of the automaton: if at least one signal y defined by the automaton has an active clock $\hat{y}$, the automaton is activated to compute it and to possibly perform some transition.

The rule $C^x_{\mathsf{init}\, s}$ defines x initially by the initial state s and then by the previous value of the next state x'. The rule $C^x_{c \Rightarrow s \to t}$ defines the next state x' by t if the current state s is x and the condition c holds. The rule $C^x_{s:p}$ defines a mode s by guarding the graph of the process p by the condition $x = s$. The term $(x = s)$ is the clock that corresponds to the condition of x being equal to s a state of the automaton. To be precise, it stands for the clock y defined by the equality test $y = \mathsf{eq}\,(x, s)$.

$$
\begin{array}{rcl}
C_a & \stackrel{\text{def}}{=} & \left(C^x_a \,\|\, (\hat{x} = \hat{x}') \,\|\, (\hat{x} = e_x)\right) / x x' \quad \text{with} \quad e_x \stackrel{\text{def}}{=} \bigvee_{y \in \text{defs}(a)} \hat{y} \\
C^x_{\mathsf{init}\, s} & \stackrel{\text{def}}{=} & \hat{x} \Rightarrow x = x' \,\mathsf{pre}\, s \\
C^x_{s:p} & \stackrel{\text{def}}{=} & (x = s) \Rightarrow G_p \\
C^x_{c \Rightarrow s \to t} & \stackrel{\text{def}}{=} & (x = s) \wedge c \Rightarrow x' = t \\
C^x_{a \,|\, b} & \stackrel{\text{def}}{=} & C_a \,|\, C_b
\end{array}
$$

The notation $(x = s) \Rightarrow G_p$ conditions the data-flow graph G_p of the process p, that models the behavior of the automaton in mode s, by the clock $(x = s)$. This means that each and every relation in G_p is then further conditioned by that clock. We write:

$$
\begin{array}{rcl}
c \Rightarrow (G \,|\, H) & \stackrel{\text{def}}{=} & (c \Rightarrow G) \,|\, (c \Rightarrow H) \\
c \Rightarrow (G / x) & \stackrel{\text{def}}{=} & (c \Rightarrow G) / x, \;\; x \notin \text{vars}(c) \\
c \Rightarrow (\hat{x} = e) & \stackrel{\text{def}}{=} & (c \wedge \hat{x}) = (c \wedge e) \\
c \Rightarrow (d \Rightarrow x = f(y_{1..n})) & \stackrel{\text{def}}{=} & (c \wedge d) \Rightarrow x = f(y_{1..n})
\end{array}
$$

5.2 Extensions and variants

While the model of mode automata we present is primarily designed to be simple, it supports extensions and variants that are of equal simplicity to formalize, thanks to the adequate intermediate representation under consideration.

Strong versus weak preemption While the presentation put the emphasis on weak preemption to model mode transitions, strong preemption can be specified and transformed into the core meta-model of Signal with an equal simplicity. Let us note $c \Rightarrow s \twoheadrightarrow t$ that immediately performs a transition from mode s to mode t upon some condition materialized by the clock c (most likely a condition on input signals such as an alarm). Then the only item to be modified in our translation scheme is the default rule for definition of the present mode x (the rule associated to $\mathsf{init}\, s_0$). It is defined by the previous value of the next state x' unless one of the conditions c of strongly preemptive transitions prevail. The effect of a strongly preemptive transition $c \Rightarrow s \twoheadrightarrow t$ is to define the current state x by t when the condition c holds and when entering in state s (i.e. when the previous value of the next state x' is s).

$$\begin{array}{rcl} C^x_{\mathsf{init}\, s_0} & \stackrel{\text{def}}{=} & \hat{x} \setminus f_x \Rightarrow x = x' \,\mathsf{pre}\, s_0 \quad \text{where} \quad f_x = \bigvee_{(c \Rightarrow s \twoheadrightarrow p) \in a} c \\ C^x_{c \Rightarrow s \twoheadrightarrow t} & \stackrel{\text{def}}{=} & ((x' \,\mathsf{pre}\, s_0) = s) \wedge c \Rightarrow x = t \end{array}$$

Reseting versus history When a mode automaton performs a transition from a mode to another, it always resets the local state of the target mode: all delayed signals defined locally in that mode are defined by their initially value while the automaton enters that mode (see Section 4.3). It may be desirable to extend this default behavior with a notion of history similar to that found in StateCharts, for instance. This can easily be done, by associating every locally delayed signal $x = y \,\mathsf{pre}\, v$ with a default assignment i that is read when the mode is entered (name this clock c) and written when the mode is left (name this clock d).

$$x = y \,\mathsf{pre}\, v \text{ becomes } (x = (z \,\mathsf{pre}\, v) \,\mathsf{when}\, c \,\mathsf{default}\, (y \,\mathsf{pre}\, v) \,|\, z = x \,\mathsf{when}\, d) \,/ z$$

Hierarchy and locality While the meta-model inherently supports notions of hierarchy and locality inherited from the GME elements, this aspect is not displayed in the formal presentation of the core model. Hierarchization is here the capability to syntactically structure an automaton into sub-automata, each of them corresponding to a particular mode, or embed mode automata into a Signal process. It is rather simple to envisage the semantics of such an extension. Should a mode be described by a mode-sub-automaton a, then its meaning would simply be that of the process C_a. This is not surprising, as hierarchical descriptions in StateCharts are mostly a syntactically structuring mechanism.

5.3 Implementation in GME

GME offers different means to extend its environment with tools, such as the *MetaGME Interpreter*, which is a plug-in accessible via the GME User Interface while modeling with MetaGME. This tool first checks the correctness of the meta-model, generates the paradigm file, and registers it into GME. This file is then used by GME to configure its environment for the newly defined paradigm.

```
1. process Switch =
2.    { type DATA_TYPE; }
3.    ( ? DATA_TYPE y1, y2; event r;
4.      ! DATA_TYPE x1, x2; )
5.    (| min_clock(x1)  |  min_clock(x2)
6.     | SwitchAtm::(| _SwitchAtm_0_currentState ^= (y1 ^+ y2 ^+ r)
7.                  | _SwitchAtm_0_reinit := ^0
8.                  | _SwitchAtm_0_nextState := (#flip when _SwitchAtm_0_reinit)
9.                                          default (#flop when (r)) when (_SwitchAtm_0_currentState = #flip)
10.                                         default (#flip when (r)) when (_SwitchAtm_0_currentState = #flop)
11.                                         default _SwitchAtm_0_currentState
12.                 | _SwitchAtm_0_currentState := _SwitchAtm_0_nextState$ init #flip
13.                 | case _SwitchAtm_0_currentState in
14.                   {#flip}: (| x2 ::= y2 | x1 ::= y1 |)
15.                   {#flop}: (| x1 ::= y2 | x2 ::= y1 |)
16.                   end
17.                 |)
18.                 where   type _SwitchAtm_0_type = enum(flip, flop);
19.                         _SwitchAtm_0_type _SwitchAtm_0_currentState, _SwitchAtm_0_nextState;
20.                         event _SwitchAtm_0_reinit;
21.                 end
22.    |)
23.   where   label SwitchAtm;
24.   end; % process Switch %
```

Figure 5: The code generated from the `switch` Model by the SIGNAL Interpreter

In a similar way as the MetaGME Interpreter, we developed an *Interpreter* to analyze Signal-Meta graphical models and produce the corresponding SIGNAL programs. This interpreter is written in C++ using the *Builder Object Network* API (BON2) provided with GME. We then extended this interpreter to produce the SIGNAL equations corresponding to mode automata descriptions. The code in Fig. 5 illustrates how the interpreter works on the switch example, which is specified in Fig. 2, 3, and 4. The transformation works as follows. For each automaton,

- one enumeration type is built (line 18). Each value of the enumeration is the name of a state (the uniqueness of names is checked).
- three signals are created: two signals for the current and next state of the automaton, which use the type built at the previous step (line 19), and one event for the reinitialization of the automaton (line 20).
- The `currentState` is defined by the value of `nextState` at the previous instant, and as the initial state for the first instant (line 12).

- The `nextState` signal is first defined by the initial state if the Automaton is reinitialized (line 8) and then, for each transition, by the destination state if the transition guard is true and if `currentState` is equal to the source of the transition (line 9-10). Finally, it is defined by the `currentState` of the Automaton (line 11). Note that the transitions are ordered according to their priority only for a state that has several output transitions.

- The `reinit` event is present when the `reinit` event of the upper level (for hierarchical Automaton) is present or when the `nextState` and the `currentState` of the upper level are different. In Fig. 5, such a definition does not appear because there are no hierarchical Automaton in the switch example. For the highest level of the automaton, the `reinit` is always absent (line 7).

- Mode changes are expressed according to the value of `currentState` (line 13-16).

In a given Automaton, the clock of `currentState` is synchronized to that of `nextState`. Nonetheless, it may be defined by that of another Automata. At the top-level, the clock of `currentState` is synchronized (line 6) only if there is some explicit synchronization in the Model, such as the Connection to `SwitchAtm` on the right of Fig. 4.

For AndStates, the interpreter only has to compose the equations of all sub-states. Finally, for States, equations are produced as for any Signal-Meta Models, which inherit from *ModelsWithDataflow* Model [18].

6 Conclusions

We have presented a model of multi-clocked mode automata defined by extending the meta-model of the synchronous data-flow specification formalism Signal in the tool GME. A salient feature of our presentation is the simplicity incurred by the separation of concerns between data-flow (that expresses structure) and control-flow (that expresses a timing model) that is characteristic to the design methodology of Signal.

While the specification of mode automata in related works requires a primary address on the semantics and on compilation of control, the use of Signal as a foundation allows to waive this specific issue to its analysis and code generation engine Polychrony and clearly expose the semantics and transformation of mode automata in a much simpler way by making use of clearly separated concerns expressed by guarded commands (data-flow relations) and by clock equations (control-flow relations).

7 Acknowledgments

The authors would like to thank Thierry Gautier for his interesting remarks on this paper.

References

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. IEEE, vol.79(9), pages 1305-1320. Septembre, 1991.

[2] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In Proc. of the IEEE *Workshop on Intelligent Signal Processing (WISP?01)*, May 2001.

[3] F. Maraninchi, and Y. Rémond, Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming, v. 46(3). Elsevier, 2003.

[4] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. POLYCHRONY for system design. *Journal of Circuits, Systems and Computers*. World Scientific, 2003.

[5] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specification. In *Application of Concurrency to System Design*. IEEE Press, 2005.

[6] J.-P. Talpin, D. Potop-Butucaru, J. Ouy, B. Caillaud. From multi-clocked synchronous processes to latency-insensitive modules. In Proc. of the *5th ACM international conference on Embedded software 2005*, Jersey City.

[7] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL V4: reference manual. `http://www.irisa.fr/espresso/Polychrony/doc_V4.15.7/document/V4_def.pdf`

[8] ISIS, Vanderbilt University. The GME Website. `http://www.isis.vanderbilt.edu/Projects/gme`

[9] ISIS, Vanderbilt University. GME User Manual. `http://www.isis.vanderbilt.edu/Projects/gme/GMEUMan.pdf`

[10] J. Bézivin, C. Brunette, R. Chevrel, F. Jouault, and I. Kurtev Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF) 4th Workshop on Best Practices for Model Driven Software Development, OOPSLA, San Diego, 2005.

[11] Potop, D., Caillaud, B. Correct-by-construction asynchronous implementation of modular synchronous specifications. In Applications of Concurrency to System Design. IEEE Press, 2005.

[12] Colaco, J. L., Girault, A., Hamon, G., Pouzet, M. Towards a higher-order synchronous dataflow language. In Embedded Software Conference, Springer Verlag lectures notes in computer science, 2004.

[13] J. T. Buck, S. Ha, E. A. Lee and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In International Journal of Computer Simulation, special issue on Simulation Software Development. v. 4, pp. 155-182. Ablex, 1994.

[14] Colaco, J.-L., Pagano, B., Pouzet, M. A conservative extension of synchronous data-flow with state machines. In Embedded Software Conference. ACM Press, 2005.

[15] D. Harel. Statecharts: a visual approach to complex systems. Science of Computer Programming, v. 8(3). Elsevier, 1987.

[16] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In Computational Engineering in Systems Applications. IMACS-IEEE, 1996.

[17] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming, v. 19(2). Elsevier, 1992.

[18] Gamatié, A., Brunette, C., Delamare, R., Gauthier, T., Talpin, J.-P. A Modeling Paradigm for Integrated Modular Avionics Design. *Submitted for publication.* INRIA, 2005.

[19] E. Rutten, F. Martinez. Signal GTI: implementing task preemption and time intervals in the synchronous data flow language Signal. In Euromicro Workshop on Real-Time Systems. IEEE Press, 1995.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399